Basics of using the readline library (https://eli.thegreenplace.net/2016/basics-of-using-the-readline-library/)

December 06, 2016 at 05:08 **Tags** C & C++ (https://eli.thegreenplace.net/tag/c-c) , Python (https://eli.thegreenplace.net/tag/python)

Did it ever happen to you to find a program that provides a shell / REPL-like interface that doesn't support basic features like line editing, command history and completion? I personally find it pretty annoying. And there's really no justification for it these days, at least on Unix-y systems, since a portable library to provide this service has existed for decades. I'm talking about `readline`.

This brief post shows some basic examples of using readline in both C++ and Python. Admittedly, it doesn't have a lot of content. Rather, its main goal is to present the accompanying code samples (https://github.com/eliben/code-for-blog/tree/master/2016/readline-samples), which I think folks will find more useful.

Let's start with a very basic readline usage example that records command history and lets us search and scroll through it:

```
#include <stdio.h>
#include <stdlib.h>
#include <string>

#include <readline/readline.h>
#include <readline/history.h>

int main(int argc, char** argv) {
  printf("Welcome! You can exit by pressing Ctrl+C at any time...\n");

  if (argc > 1 && std::string(argv[1]) == "-d") {
    // By default readline does filename completion. With -d, we disable this
    // by asking readline to just insert the TAB character itself.
    rl_bind_key('\t', rl_insert);
  }

  char* buf;
  while ((buf = readline(">> ")) != nullptr) {
    if (strlen(buf) > 0) {
      add_history(buf);
    }

    printf("[%s]\n", buf);

    // readline malloc's a new buffer every time.
    free(buf);
  }

  return 0;
}
```

The main thing in this code sample is using the `readline` function instead of standard language APIs for user input (such as `fgets` in C or `std::getline` in C++). This already gives us many of readline's features like line editing: having typed a word, we can actually go back and fix some part of it (using the left arrow key or `Ctrl-B`), jump to input end (`Ctrl+E`) and so on - all the editing facilities we're so used to from the standard Linux terminal.

The `add_history` calls go further: they add every command typed into the history buffer. With this done, we can now scroll through command history with up/down arrows, and even do history seaches with `Ctrl+R`.

Note also that readline automatically enables tab completion. The default completion functionality auto-completes file names in the current directory, which isn't something we necessarily want. In this sample, tab completion is optionally disabled by binding the tab key to `rl_insert`, which just sends the actual key code to the terminal rather than doing anything special like completion.

Simple completion

Implementing custom completion with readline is fairly simple. Here is a sample (https://github.com/eliben/code-for-blog/blob/master/2016/readline-samples/readline-complete-simple.cpp) that will tab-complete words from a certain vocabulary. The `main` function remains as before, with a small difference - registering our completion function with readline.

```
rl_attempted_completion_function = completer;
```

Configuring readline happens through global variables it exports. These variables are all documented (http://www.delorie.com/gnu/docs/readline/rlman_47.html). While we could use `rl_completion_entry_function` to make our code slightly shorter, for extra fun let's instead use `rl_attempted_completion_function` - it lets us customize things a bit more. The default `rl_completion_entry_function` performs filename completion in the current directory. We can disable it in our own "attempted" completion function:

```
char** completer(const char* text, int start, int end) {
  // Don't do filename completion even if our generator finds no matches.
  rl_attempted_completion_over = 1;

  // Note: returning nullptr here will make readline use the default filename
  // completer.
  return rl_completion_matches(text, completion_generator);
}
```

Otherwise, it's all the same. We have to implement a "completion generator" and pass it to the helper `rl_completion_matches` to generate the actual matches. Our completion generator auto-completes from a global vocabulary of words:

```
char* completion_generator(const char* text, int state) {
  // This function is called with state=0 the first time; subsequent calls are
  // with a nonzero state. state=0 can be used to perform one-time
  // initialization for this completion session.
  static std::vector<std::string> matches;
  static size_t match_index = 0;

  if (state == 0) {
    // During initialization, compute the actual matches for 'text' and keep
    // them in a static vector.
    matches.clear();
    match_index = 0;

    // Collect a vector of matches: vocabulary words that begin with text.
    std::string textstr = std::string(text);
    for (auto word : vocabulary) {
      if (word.size() >= textstr.size() &&
          word.compare(0, textstr.size(), textstr) == 0) {
        matches.push_back(word);
      }
    }
  }

  if (match_index >= matches.size()) {
    // We return nullptr to notify the caller no more matches are available.
    return nullptr;
  } else {
    // Return a malloc'd char* for the match. The caller frees it.
    return strdup(matches[match_index++].c_str());
  }
}
```

You can read more details about how the completion works on this page (http://www.delorie.com/gnu/docs/readline/rlman_45.html). The samples respository (https://github.com/eliben/code-for-blog/tree/master/2016/readline-samples) contains several additional variations on this theme, including a more sophisticated program that provides hierarchical completion of sub-commands, where the first token determines the autocompletion vocabulary for subsequent tokens.

Python

The Python standard library comes with a `readline` module that provides an intereface to the underlying C library. In fact, it can also use `libedit` under the hood. `libedit` is the BSD implementation of the `readline` interface, and can be used on some platforms. In Python you don't have to care about this though.

A basic completion example in Python using `readline` is as simple as:

```
import readline

def make_completer(vocabulary):
    def custom_complete(text, state):
        # None is returned for the end of the completion session.
        results = [x for x in vocabulary if x.startswith(text)] + [None]
        # A space is added to the completion since the Python readline doesn't
        # do this on its own. When a word is fully completed we want to mimic
        # the default readline library behavior of adding a space after it.
        return results[state] + " "
    return custom_complete

def main():
    vocabulary = {'cat', 'dog', 'canary', 'cow', 'hamster'}
    readline.parse_and_bind('tab: complete')
    readline.set_completer(make_completer(vocabulary))

    try:
        while True:
            s = input('>> ').strip()
            print('[{0}]'.format(s))
    except (EOFError, KeyboardInterrupt) as e:
        print('\nShutting down...')

if __name__ == '__main__':
    main()
```

It's fairly obvious that the Python API is a thin veneer around the underlying C API - the completion implements state in the same way. That said, Python's built-in features like first order functions and lexical closures make writing more sophisticated completion code a much simpler task. See the included code samples (https://github.com/eliben/code-for-blog/tree/master/2016/readline-samples/python) for more examples.

Other libraries and modules

Frustrated by the complexity of readline, Salvatore Sanfilippo (of Redis fame) created a simple line-completion library named linenoise (https://github.com/antirez/linenoise). It has a prety simple interface and is very small, so I think it's good for inclusion into projects that want to minimize system dependencies (like figuring out how to link with readline on different platforms).

On the Python side, there are a couple of other related modules in the standard library I'd like to mention. One is rlcompleter (https://docs.python.org/dev/library/rlcompleter.html); think of it as readline pre-configured with completions for Python functions and identifiers. If you want to build Python interpreters or

shells, this is the tool for you.

Yet another Python module is cmd (https://docs.python.org/dev/library/cmd.html); it's a higher-level abstraction around readline, allowing one to encapsulate commands into class methods. If you want to write a simple command interpreter (think Logo), using `cmd` will result in less code.

IMHO while having `readline` in the standard Python library is wonderful , both `rlcompleter` and `cmd` are signs that the Python developers sometimes go too far with the "batteries included" philosophy. But YMMV.

For comments, please send me an email (mailto:eliben@gmail.com), or reach out on Twitter (https://twitter.com/elibendersky).

Back to top